e-Informatica Software Engineering Journal
Volume 21, Issue 1, pages: 1–33
DOI: 10.37190/e-Inf270102
BibTeX

Text-to-SQL Dataset Quality Assessment: A Multi-Dimensional Validation Framework

Volodymyr Sabadosh*, Vladyslav Kotsovsky
*Corresponding author: volodymyr.sabadosh@uzhnu.edu.ua

Article info

Dataset link:
https://github.com/vsabadosh/text2sql-dataset-analyzer

Keywords:
Text-to-SQL system
dataset quality
LLM
SQL validation
semantic evaluation
empirical software engineering
benchmark auditing
LLM-as-a-judge

Submitted: 08 Mar. 2026
Revised: 16 Aug. 2026
Accepted: 18 Aug. 2026
Available online: 28 Aug. 2026

Abstract

Context: LLM-based Text-to-SQL has advanced quickly, but benchmark and training datasets may contain defects that distort evaluation and fine-tuning. Prior audits remain fragmented, addressing dimensions in isolation.
Objective: This paper proposes `text2sql-dataset-analyzer`, an open-source framework that audits Text-to-SQL datasets across five complementary quality dimensions within a single reproducible pipeline.
Method: The framework covers five dimensions: database schema integrity, SQL syntactic structure and complexity, execution testing, antipattern detection, and semantic correspondence. Semantic correspondence is evaluated via an LLM-as-a-judge committee with majority voting. An analytical database stores the resulting metrics for direct querying and Markdown report generation, while structured JSONL output records per-item annotations.
Results: Auditing all 11,840 Spider 1.0 examples reveals quality issues despite 99.97% of queries passing execution checks. Schema and data checks identify 51 structural foreign-key errors and 41,927 row-level referential-integrity violations, 41,913 of them in just three of 206 databases. At the item level, the audit flags unanimously Incorrect (8–10%), disputed (23–27%), and Unanswerable NL–SQL pairs, together with SQL antipatterns associated with correctness, robustness, and portability concerns. Manual review of 367 flagged items confirms genuine defects in 263, including 26 consensus Unanswerable questions and all 17 Cartesian-product join bugs.
Conclusions: Multi-dimensional validation exposes dataset defects missed by executability-based checks. We release the open-source framework and a prioritized remediation roadmap for a popular Text-to-SQL benchmark. The downstream impact of these defects on model training and benchmark scores remains future work.

1. Introduction

Text-to-SQL systems, which transform natural language queries into structured SQL queries, are experiencing a period of rapid development driven primarily by remarkable progress in Large Language Models (LLMs), as well as fundamental scientific interest and practical industry needs.

From a scientific perspective, the Text-to-SQL task serves as an important test of LLMs' capacity for structured reasoning and simultaneous understanding of natural language

© 2027 The Authors. Published by Wrocław University of Science and Technology Publishing House.
This is an open access article under the CC BY-SA 4.0 license international.

semantics and formal programming languages [1, 2]. Recent GPT-4-based systems have reached 86.6% execution accuracy [3] on the Spider 1.0 leaderboard [4] and 80.88% [5] on the BIRD leaderboard [6]. Reported Spider test exact-match scores were 66.0% for DAIL-SQL+GPT-4 with self-consistency, compared with 4.8–12.4% for 2018 non-LLM baselines (Seq2Seq+Attention, SQLNet, TypeSQL) [3, 7].

From a practical standpoint, Text-to-SQL LLM systems are being actively deployed by leading cloud platforms. Snowflake Copilot can generate SQL from natural language in Snowsight [8]; Databricks AI/BI Genie converts plain English questions into equivalent SQL queries [9]; Microsoft Copilot in Fabric/Azure SQL supports Natural Language → SQL in SQL Database workspaces [10]. This lowers the barrier for business analysts and non-technical users, providing data access without knowledge of SQL syntax.

The typical process of training LLMs for Text-to-SQL tasks includes three key components (Figure 1): a training dataset containing triplets of (natural language question, database schema, target SQL query), an adaptation method (fine-tuning or few-shot learning) and evaluation (content matching and execution-based). During fine-tuning, models adjust their weights on large Text-to-SQL datasets to better capture the mapping between natural language and SQL. In few-shot settings, by contrast, the model relies on in-context learning, using only a handful of examples in the prompt without updating its parameters. In both cases, dataset quality is critical: in fine-tuning, even a small fraction of flawed examples among tens of thousands can introduce systematic errors, while in few-shot prompting each individual example directly shapes model behavior, so a single incorrect pair may lead to consistently wrong generations (see Section 2.2). The following types of errors may be present in datasets:

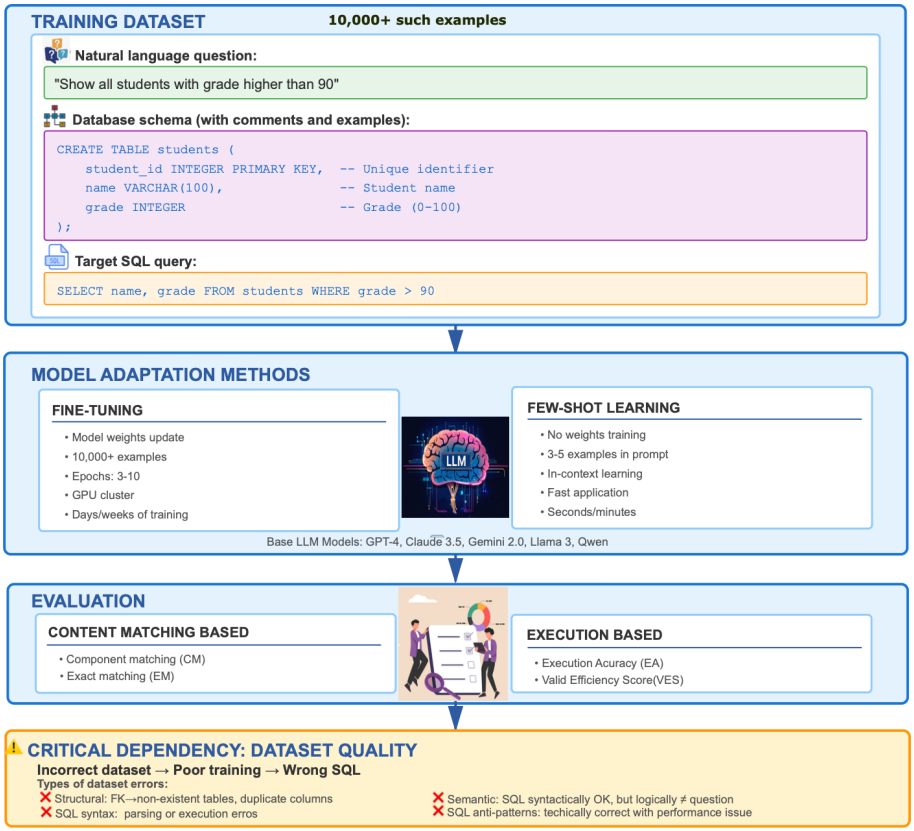

Figure 1. Complete training pipeline for LLMs in Text-to-SQL systems

- **Database schema structural errors:** incorrect foreign keys (references to non-existent tables/columns), duplicate columns, and multiple primary keys.
- **SQL syntax/execution errors:** queries that cannot be parsed or executed on corresponding databases.
- **Semantic mismatches:** SQL queries that are syntactically correct but logically do not correspond to the natural language question (e.g., using SUM instead of AVG, incorrect JOIN conditions, missing necessary filters).
- **SQL antipatterns:** technically correct queries with performance or style issues (SELECT *, absence of LIMIT, functions in WHERE that block index usage).

Problem statement

Despite the central role of datasets, most work on Text-to-SQL focuses on model architectures, prompting techniques and decoding strategies. Until recently, the quality of popular benchmarks was typically assumed rather than systematically measured. Recent studies have begun to scrutinize benchmark quality more directly, but existing studies either provide high-level descriptive statistics, focus on semantic label auditing for selected subsets, or concentrate on runtime validation of single generated queries rather than auditing entire corpora (see Section 2 for a detailed discussion). Consequently, researchers and practitioners still have limited visibility into structural defects, semantic mismatches, or stylistic antipatterns hidden in widely used Text-to-SQL datasets. Therefore, our central research questions are:

- **RQ1:** How can we systematically evaluate the overall quality of Text-to-SQL datasets?
- **RQ2:** How accurate and internally consistent are popular Text-to-SQL benchmarks that serve simultaneously as training and evaluation data?

Research objective

This paper addresses these questions by proposing a multi-dimensional validation framework for Text-to-SQL datasets and using Spider 1.0 as the primary end-to-end testbed for the framework. Our work is positioned within empirical software engineering: we treat benchmark datasets as software artifacts that should be inspected, measured and improved with the same rigor as source code or test suites.

Contribution statement

The main contributions of this paper are:

1. A multi-layer validation architecture for Text-to-SQL datasets that integrates DDL schema validation, syntax analysis, execution testing, antipattern detection, and LLM-based assessment of NL–SQL semantic correspondence.
2. An open-source implementation of the validation framework, publicly available to enable reproducible quality assessment of existing and future Text-to-SQL benchmarks.
3. The first empirical audit of all three Spider 1.0 partitions (11,840 examples in total) across five complementary quality dimensions.
4. Concrete empirical evidence that widely-used benchmarks contain significant quality issues invisible to standard evaluation metrics, including highly concentrated referential-integrity violations, substantial schema heterogeneity, and systematic semantic ambiguities.

2. Related work

We review prior work in three areas: Text-to-SQL benchmarks and datasets, data quality and robustness in Text-to-SQL, and dataset quality assessment frameworks.

2.1. Text-to-SQL benchmarks and datasets

Text-to-SQL benchmarks have shaped the field's modeling and evaluation conventions. WikiSQL [11] introduced 80,654 NL–SQL pairs over 24,241 Wikipedia tables and was one of the first large-scale benchmarks for Text-to-SQL. Its design allowed only single-table queries, excluding JOINs, nested subqueries, and complex aggregations.

Spider [7] addressed these limits by introducing a cross-domain benchmark with 10,181 natural-language questions and 5,693 unique SQL queries across 200 databases in 138 domains. Its strengths include multi-table schemas with foreign keys, balanced difficulty distribution, and broad SQL coverage spanning nested queries, set operations, and grouped aggregations. Spider remains one of the most widely used Text-to-SQL benchmarks and continues to serve as a standard reference point for cross-domain evaluation.

BIRD [12] complements Spider with content-rich, large-scale databases of approximately 33 GB total. It contains 12,751 question–SQL pairs across 95 substantial databases in 37 professional domains. BIRD adds external knowledge dependencies and noisy real-world values, targeting robustness under production-like conditions.

Synthetic corpora extend dataset scale further. Gretel-Synth [13] provides over 100,000 examples across business domains. OmniSQL [14] reaches over 2.5 million instances across 16,000+ synthetic databases with chain-of-thought annotations.

The growth of both human-curated and synthetic resources makes corpus-level quality control increasingly important. Larger datasets make manual inspection infeasible at scale, motivating automated, multi-dimensional quality assessment of Text-to-SQL corpora.

2.2. Data quality and robustness in Text-to-SQL

Empirical evidence shows that controlled variations in benchmark data directly affect model performance and evaluation outcomes. Dr.Spider [15] applies 17 perturbations to databases, natural-language questions, and SQL queries. The most robust model, PICARD, shows a 14.0% relative drop in macro-averaged execution accuracy overall and a 50.7% drop under the DBcontent-equivalence perturbation. These results indicate that current models are sensitive to variations in data representation and query semantics.

Snowflake's Arctic-Text2SQL study [16] shows that the inclusion of lower-quality synthetic data, such as Gretel-Synth, can reduce model accuracy despite increasing the size of the training set. Their experiments suggest that data quality is more important than dataset volume for effective Text-to-SQL training.

Jin et al. [17] provide direct evidence that annotation repair can change model comparisons. They report annotation-error rates of 52.8% on BIRD Mini-Dev and 62.8% on Spider 2.0-Snow. After correcting a 100-item BIRD Dev subset, they re-evaluate 16 open-source agents from the BIRD leaderboard. Relative performance changes range from -7% to $+31\%$, ranking shifts span -9 to $+9$ positions, and the Spearman rank correlation between original and corrected leaderboards drops from 0.85 to 0.32.

2.3. Dataset quality assessment frameworks

Auditing approaches in the literature differ substantially in scope. Mitsopoulou and Koutrika [18] characterize Text-to-SQL datasets along multiple dimensions covering SQL-query structure, operator usage, database-schema properties, and lexical or syntactic properties of natural-language questions. They also propose an "informative evaluation" procedure that decomposes model errors into structural, operator, and variable mismatches. Their framework is well suited to aggregate dataset characterization and fine-grained system evaluation, but it is not designed to audit the correctness of individual dataset annotations.

Pandey et al. [19] focus instead on runtime validation for enterprise Text-to-SQL systems. Their framework covers several validation stages, including query-construction validation, data-integrity verification, automated feedback generation, and error detection and correction. These dimensions are closely related to our objectives, but the work is primarily conceptual and targets per-query validation in deployed systems rather than dataset-level auditing of benchmark annotations. It also does not provide an open-source implementation.

Recent work has also begun to examine the semantic correctness of gold Text-to-SQL annotations. Jin et al. [17] proposed SAR-Agent, an LLM-based multi-turn reviewer that interacts with target databases and generates diagnostic reports for annotated examples. They also introduce SAPAR, a correction pipeline that integrates SAR-Agent into the annotation workflow. However, the released artifact provides dataset-specific entry points for BIRD and Spider 2.0-Snow rather than a configuration-driven auditing framework, and its scope is limited to the semantic correctness of gold annotations.

NL2SQL-BUGs [20] provides a complementary resource: an expert-annotated corpus of 2,018 NL–SQL pairs, including 999 examples with semantic bugs organized under a two-level taxonomy. The authors use GPT-4o as an automatic semantic-error detector on subsets of Spider and BIRD, and confirm previously unknown errors through manual review. The public release includes the labeled benchmark, taxonomy, and prompt templates, but not an executable detection pipeline. As a result, it mainly supports manual semantic review rather than end-to-end corpus auditing.

Methodologically, semantic auditing connects to the broader LLM-as-a-judge literature. Zheng et al. [21] showed that strong LLM judges can approximate human preference judgments, while also exhibiting risks such as position bias, verbosity bias, self-enhancement bias, and limited reasoning ability. ChatEval [22] further shows that committee-style and deliberation-based evaluation can reduce reliance on a single judge and expose disagreement among evaluators. For Text-to-SQL auditing, these results motivate treating LLM verdicts as structured semantic signals that require controlled prompts, multi-model consensus, and human calibration, rather than as unqualified ground truth.

SQL-quality analysis forms a separate database-systems lineage. Karwin's catalog [23] systematizes recurring SQL and database-design antipatterns, while SQLCheck [24] implements rule-based checks for common query-level antipatterns based on this taxonomy. SQLCheck is relevant to our work as a SQL-only baseline, but its scope differs from Text-to-SQL corpus auditing: it targets individual SQL queries rather than dataset-level quality analysis. Moreover, its rule set covers only part of the pattern space relevant to Text-to-SQL benchmark auditing, leaving out cases such as unsafe data-modification queries, order-sensitive pagination, and null-sensitive subquery patterns.

Schema-level validation has also been studied extensively in database-systems research and is supported by practical tooling. Relational data profiling methods [25] detect de-

pendencies such as inclusion dependencies, unique column combinations, and candidate foreign keys. The work [26] is especially relevant to Text-to-SQL benchmarks because they analyze foreign-key violations under SQL null semantics when foreign-key enforcement is disabled. This setting matches benchmarks such as Spider 1.0, whose SQLite databases do not enforce foreign keys by default. In practice, SchemaCrawler [27] provides schema discovery, documentation, diagrams, and lint-style checks for potential schema-design issues. Together, these works establish schema validation as a mature component of database quality assurance. However, they treat schemas and relational instances as standalone artifacts, rather than linking schema-level issues to individual Text-to-SQL dataset items.

Table 1 compares the surveyed approaches across the quality dimensions relevant to Text-to-SQL corpus auditing.

Table 1. Positioning of prior work on Text-to-SQL dataset quality auditing. ✓ = full coverage, △ = partial or indirect coverage, × = absent

Approach	Main purpose	C1	C2	C3	C4	C5	C6	Limitation for corpus auditing
Mitsopoulou and Koutrika [18]	Benchmark profiling and informative evaluation	△	△	×	×	×	△	Does not verify gold labels or produce defect annotations.
Jin et al. [17]	LLM-agent semantic annotation audit	×	△	×	✓	✓	△	Dataset-specific entry points; limited to semantic correctness.
NL2SQL-BUGs [20]	Semantic bug benchmark and taxonomy	×	×	×	✓	✓	×	No executable pipeline for full-corpus auditing.
Pandey et al. [19]	Enterprise runtime validation	△	✓	×	△	×	×	Conceptual; no open implementation or benchmark-level audit.
SQLCheck [24]	SQL antipattern detection	×	△	✓	×	△	✓	SQL-only; limited Text-to-SQL pattern coverage.
Relational FK profiling [25, 26]	FK relationship and referential integrity validation	✓	×	×	×	×	×	Does not link findings to individual NL–SQL records.
SchemaCrawler [27]	General-purpose schema discovery and lint-style inspection	✓	×	×	×	×	✓	Schema-only; no Text-to-SQL item-level audit.
This paper	Multi-dimensional Text-to-SQL corpus audit	✓	✓	✓	✓	✓	✓	

Comparison criteria: C1 – DB schema integrity; C2 – SQL parse & execute; C3 – Antipattern rules; C4 – NL↔SQL semantic check; C5 – Per-item annotations; C6 – Reusable tool.

The comparison in Table 1 highlights a persistent methodological gap. Existing Text-to-SQL studies provide valuable insights into benchmark structure, model robustness, and semantic annotation errors, while database-engineering tools offer mature mechanisms for schema inspection and SQL-quality analysis. However, no surveyed Text-to-SQL approach covers all six comparison criteria (C1–C6) in Table 1, and general-purpose database tools do not connect their findings to natural-language questions and gold SQL annotations. Consequently, widely used benchmarks such as Spider 1.0 and BIRD still lack a multi-dimensional quality analysis, despite years of active use. We close this gap by

developing a multi-dimensional open-source validation framework and performing a quality assessment of the Spider 1.0 benchmark.

3. Methods

This section describes the design, architecture, and implementation of our Text-to-SQL dataset validation framework.

3.1. Architectural overview of the text2sql-dataset-analyzer framework

We introduce `text2sql-dataset-analyzer`, a novel framework and, to the best of our knowledge, the first dedicated tool for systematic, multi-dimensional quality assessment of Text-to-SQL benchmarks. The framework is released as open-source software, which allows other researchers and practitioners to inspect, extend, and apply it to their own Text-to-SQL corpora.

Design principles. The framework uses a streaming, configuration-driven architecture for large-scale Text-to-SQL dataset validation and quality assessment. It follows three design principles: modular component interfaces, constant-memory processing, and extensibility through configuration-driven composition. This design enables the framework to process large datasets while maintaining a deterministic memory footprint, which is important when analyzing benchmarks ranging from Spider 1.0 to multi-gigabyte datasets such as BIRD or OmniSQL 2.5 M.

Technology stack. The framework is implemented in Python 3.11 [28] and combines SQLGlot [29] for dialect-aware SQL parsing, SQLAlchemy 2.0 [30] for portable database access (currently SQLite and PostgreSQL), and DuckDB [31] as the columnar analytics backend for metric storage and querying. Pipeline composition is configuration-driven via YAML and dependency injection [32, 33].

3.2. Five-layer system architecture

The framework architecture follows a layered design pattern comprising five runtime layers, all orchestrated by a separate configuration and orchestration layer that declaratively specifies components and their parameters (Figure 2). This layered organization promotes separation of concerns, facilitates independent testing and development of components, and enables selective activation or deactivation of functionality based on analysis requirements.

Configuration and orchestration. A cross-cutting YAML configuration layer declaratively specifies all loaders, normalizers, analyzers, and output formats. A dependency-injection container instantiates and wires the corresponding components at runtime.

Input layer. Streaming loaders abstract heterogeneous data sources (JSONL, CSV, JSON, Hugging Face Datasets Hub) behind a common iterator interface, yielding records on demand to maintain constant memory usage regardless of dataset size.

Normalization layer. The Normalization Layer standardizes records originating from diverse datasets through three sequential normalizers. The Alias Mapper reconciles different field names (e.g., `query` vs `sql`, `db_id` vs `dbId`) into a canonical schema. The ID Assignment component guarantees a unique identifier for each record using either counters

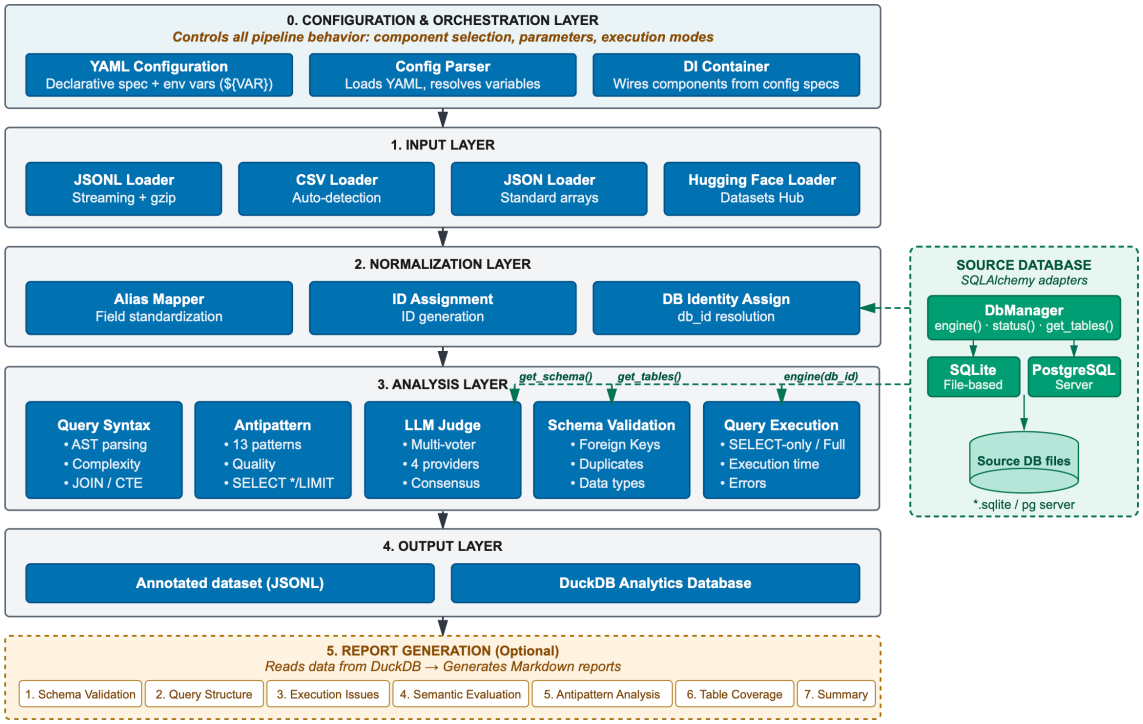

Figure 2. Five-layer architecture of the text2sql-dataset-analyzer framework

or content-based hashes, supporting consistent cross-referencing between the annotated dataset and the metrics database. The DB Identity Normalizer links each record to a valid database through a three-branch resolution strategy. When an explicit database identifier is present, it is validated via a health check against the database manager. When only a DDL schema is available (common in synthetic datasets), a deterministic SHA-256 hash of the schema text serves as the identifier, with caching to avoid redundant database creation for records sharing identical schemas. When neither identifier nor schema is provided, the normalizer raises an error, enforcing a fail-fast policy that guarantees downstream analyzers can rely on a valid database reference. The result is a normalized DataItem with mandatory fields (`id`, `question`, `sql`, `dbId`), an optional schema, and a metadata dictionary.

Analysis layer. The Analysis Layer is the computational core, hosting five analyzers that cover complementary quality dimensions of Text-to-SQL data:

1. **Schema validation analyzer** checks database integrity, including several classes of structural foreign-key errors and case-insensitive column name duplicates. A per-database cache reuses schema introspection results across queries.
2. **Query syntax analyzer** performs static analysis using the SQLGlot parser. It builds abstract syntax trees, extracts structural features (e.g., JOIN counts, subquery depth, CTEs, aggregations, window functions), and assigns difficulty labels (Easy, Medium, Hard, Expert) via rule-based classification.
3. **Query execution analyzer** evaluates dynamic executability within the corresponding database. Queries are executed in a sandbox according to the configured mode. The analyzer records execution time and row counts and classifies failures as syntax, semantic, or connectivity issues.
4. **Query antipattern detector** uses a rule-based approach to flag 13 SQL antipatterns, such as functions in WHERE predicates blocking index usage, correlated subqueries with quadratic complexity, and implicit joins.

5. **Semantic LLM judge analyzer** uses large language models to assess how well SQL queries answer their natural-language questions.

Output layer. Results are persisted through two parallel streams: an annotated JSONL dataset preserving per-item provenance, and a DuckDB analytics database enabling SQL-based metric querying and export without re-running the pipeline.

Report generation layer (optional). This layer converts stored DuckDB metrics from any completed analysis run into seven human-readable Markdown reports: Schema Validation, Query Structure, Execution Issues, Semantic Evaluation, Antipattern Analysis, Table Coverage, and Summary.

3.3. Schema validation analyzer

The Schema Validation Analyzer detects structural schema errors and row-level data-quality issues that could compromise query execution or semantic validity. It identifies problems in table definitions, constraint specifications, and referential integrity that affect all queries operating against those databases. The analyzer uses a dual-level validation architecture. It separates structural errors in schema declarations (design flaws that prevent proper constraint enforcement) from data violations in existing rows (integrity breaches where stored data contradicts declared constraints). This separation enables both proactive schema repair and targeted data cleaning strategies.

Figure 3 illustrates the three-branch schema validation architecture. The analyzer processes databases through parallel validation pathways, each targeting distinct quality dimensions. The first branch extracts foreign key definitions from database metadata through dialect-specific system catalogs, with fallback to dataset-provided schema files when explicit constraint metadata proves unavailable. The second branch performs schema introspection via column enumeration and primary key discovery to detect structural anomalies. The third branch executes data queries to assess content quality. This multi-pathway design enables validation spanning constraint integrity, column definitions, and data completeness.

Foreign key structural validation examines constraint definitions through four checks: Missing Parent Table (verifying referenced tables exist), Missing Parent Column (confirming parent column presence), Arity Mismatch (enforcing equal column counts in composite

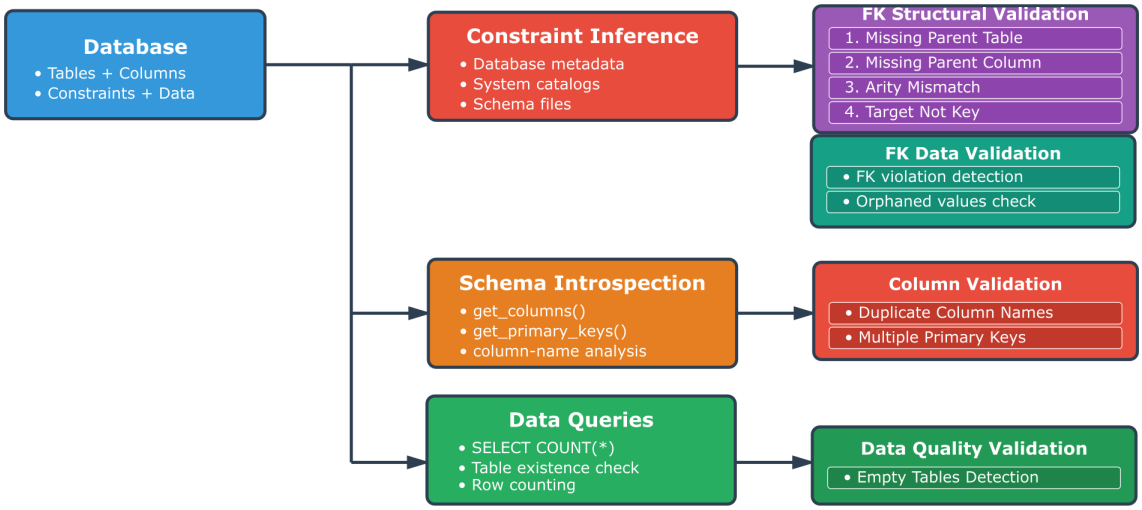

Figure 3. Schema validation analyzer architecture

keys), and Target Not Key (verifying parent columns possess uniqueness constraints). Foreign key data validation detects orphaned values through dialect-specific integrity check commands. These are rows where foreign key column values lack corresponding parent table entries. Additional schema validation detects case-insensitive duplicate column names and tables with multiple primary key definitions. Data quality validation executes row count queries to identify empty tables. Such empty tables can distort model training or produce misleading evaluation results when queries execute successfully against them.

3.4. Query syntax analyzer

The Query Syntax Analyzer performs static structural analysis of SQL statements, extracting syntactic features and classifying query difficulty without database execution. The analyzer processes queries through three stages: abstract syntax tree construction, structural feature extraction, and rule-based difficulty classification (Figure 4).

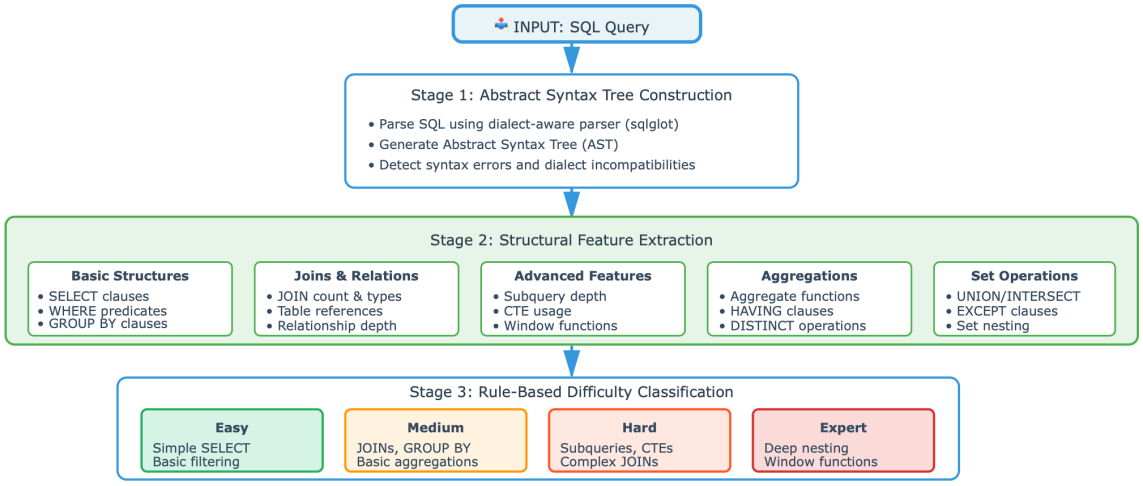

Figure 4. Query Syntax Analyzer architecture with three-stage processing pipeline

The Abstract Syntax Tree Construction stage uses the SQLGlot parser to transform SQL text into a hierarchical tree whose nodes represent syntactic constructs: SELECT expressions, JOIN operations, WHERE predicates, subqueries, and CTEs. Parsing failures are captured with detailed error messages to support identification of malformed queries.

In the Structural Feature Extraction stage, the analyzer traverses the AST to compute a feature vector capturing syntactic properties relevant to query complexity. Five feature groups are extracted. Basic structure features count projections, quantify WHERE predicate complexity, and detect GROUP BY and ORDER BY clauses. Join and relation features count JOIN operations, identify their types (INNER, LEFT,...), and enumerate distinct tables, capturing relational complexity. Advanced features measure subquery nesting depth, CTE usage, window functions (OVER clauses), and recursive CTEs. Aggregation features track aggregate functions (COUNT, SUM,...), HAVING clauses, and DISTINCT operations. Set operation features detect UNION, INTERSECT, and EXCEPT operators and their nesting patterns.

The Difficulty Classification stage applies a priority-cascading rule system that evaluates the extracted feature vector top-down, from the most complex patterns to the simplest, and assigns one of four categorical labels. Expert queries are identified first: any query containing

recursive CTEs, subquery nesting depth ≥ 2, three or more set operations, or two or more co-occurring advanced features (CTEs, window functions, multiple subqueries) is classified as Expert. Hard queries exhibit at least one advanced SQL feature: any subquery, CTE, window function, two or more set operations, four or more tables, three or more JOINs, or a HAVING clause. Medium queries introduce moderate structural elements: multi-table schemas (≥ 2 tables), at least one JOIN, GROUP BY, aggregate functions, three or more WHERE conditions, a single set operation, or three or more co-occurring simple features. Easy queries are the residual category: single-table SELECT statements with at most basic filtering and sorting. This rule-based approach adapts Spider's original four-level difficulty classification [7], which relies on counts across predefined SQL component groups, by incorporating a broader feature space that covers advanced SQL constructs (CTEs, window functions, recursive queries) absent from Spider's 2018-era scheme, and by applying explicit AST-based thresholds (e.g., subquery nesting depth, combined feature detection) beyond its component-counting criteria. The resulting classification enables benchmark coverage assessment, stratified evaluation, and difficulty imbalance identification across heterogeneous Text-to-SQL datasets.

3.5. Query antipattern detector

The Query Antipattern Detector implements a rule-based static analysis framework for identifying SQL constructs that compromise query correctness, data integrity, execution performance, or code maintainability. Antipattern detection applies domain-specific knowledge to identify patterns empirically associated with incorrect results, data corruption, or performance degradation. The detection methodology employs a two-stage analytical pipeline (Figure 5): AST-based pattern recognition through structural matching and dialect-configurable severity classification.

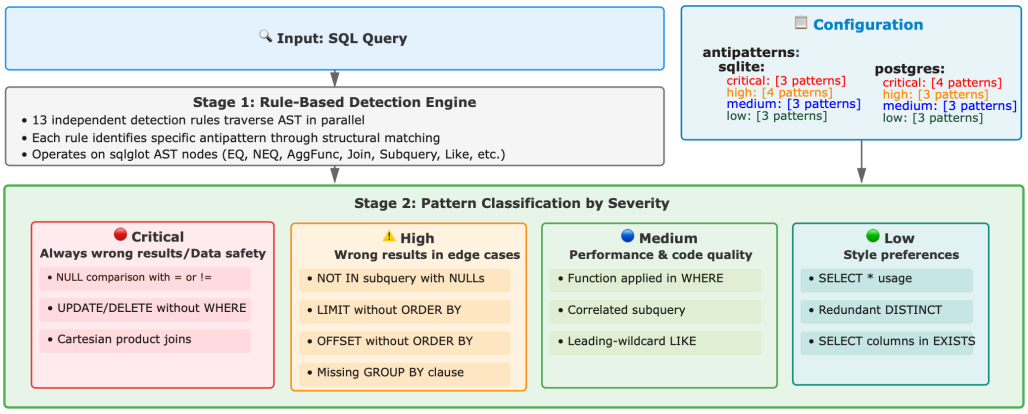

Figure 5. Query Antipattern Detector Architecture

Rule-based detection engine. The detector implements 13 independent rules for identifying common SQL antipatterns documented in established SQL literature. The rule set is designed as a practical, extensible baseline rather than an exhaustive taxonomy. Detection leverages typed AST nodes (EQ, NEQ, AggFunc, Update, Delete, Where, Join, Subquery, Like) to implement precise structural pattern matching across four quality dimensions: correctness violations, data safety issues, performance degradations, and code quality concerns.

Severity classification. The detector implements dialect-configurable severity assignment, acknowledging that antipattern impact varies across SQL implementations. Separate configuration mappings exist for SQLite and PostgreSQL. Table 2 summarizes all 13 detected antipatterns with their severity classifications and descriptions.

Table 2. Summary of 13 detected SQL antipatterns by severity (SQLite configuration)

Severity	Antipattern	Impact
Critical	NULL comparison (`col = NULL`)	Always evaluates to NULL, breaking predicate logic [34]
	Unsafe UPDATE/DELETE	Missing WHERE modifies/deletes all rows [23]
	Cartesian product joins	Missing JOIN condition causes $N \times M$ row explosion [35]
High	NOT IN with nullable subquery	NULL causes entire predicate to fail silently [34]
	LIMIT without ORDER BY	Non-deterministic result subset [36]
	OFFSET without ORDER BY	Non-deterministic pagination [36]
	Missing GROUP BY	Bare columns may produce arbitrary values [37, 38]
Medium	Functions in WHERE	Prevents index usage, forces full scan [39]
	Correlated subqueries	$O(n^2)$ row-by-row execution [35]
	Leading wildcard LIKE	Prevents index prefix matching [23]
Low	SELECT *	Obscures column dependencies [23]
	Redundant DISTINCT + GROUP BY	Unnecessary duplicate elimination [39]
	Columns in EXISTS subquery	Wasteful column retrieval [39]

It is worth noting that, although the Query Antipattern Detector is just one of several modules in the overall system, it received the most extensive test coverage. Specifically, we implemented 279 pytest test cases that exercise all 13 detection rules and their dialect-configurable severity mappings. The suite targets high-risk SQL edge cases (e.g., NULL three-valued logic, omitted join predicates, non-deterministic pagination via LIMIT/OFFSET without ORDER BY, and aggregation corner cases), providing a regression harness against rule drift as the detector evolves.

3.6. Semantic LLM judge analyzer

The Semantic LLM Judge Analyzer complements syntax and execution checks by using LLMs to assess whether each SQL query captures the intent of its associated natural-language question. Its four-stage pipeline comprises schema context synthesis, prompt engineering, multi-model judgment collection, and result aggregation (Figure 6).

Schema context synthesis. The first stage prepares a structured representation of the database schema and sample data that is injected as context into the LLM-based semantic judge. The analyzer retrieves metadata for the entire database schema (tables, columns, data types, primary keys, and foreign keys) and executes SELECT queries to sample a configurable number of example values from each column (by default, two per column), giving the language model concrete data instances that complement the type information. The collected metadata is then formatted into CREATE TABLE statements with inline comments containing the example values, producing a compact DDL specification.

Prompt template resolution. The second stage combines the generated DDL with contextual elements to produce complete evaluation prompts. Templates are loaded from YAML configuration files to facilitate experimentation with different prompt formulations. Each template contains dynamic placeholders:

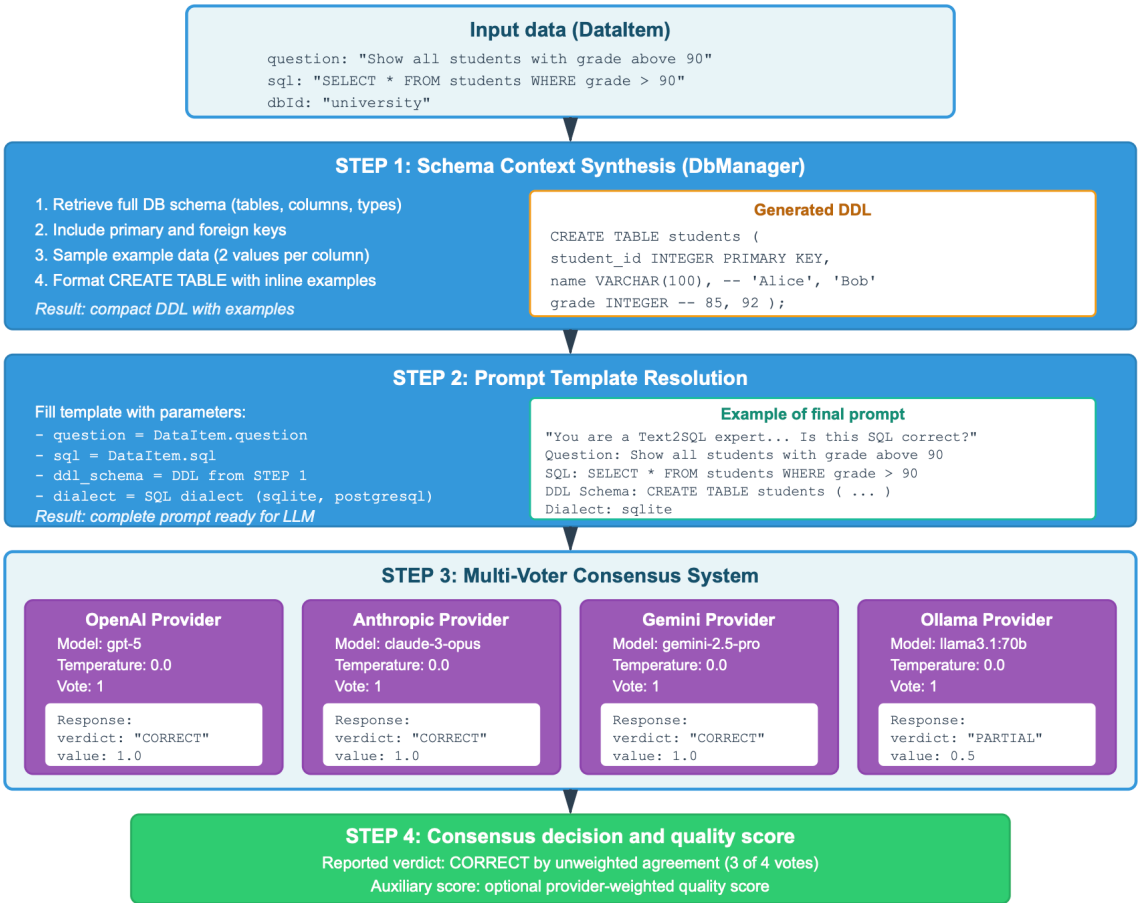

Figure 6. Architecture of the Semantic LLM Judge Analyzer with multi-model consensus voting

– `{{question}}`: The natural language question from the dataset, providing ground truth for query semantics
– `{{sql}}`: The SQL query under evaluation, formatted with syntax highlighting and normalized whitespace
– `{{ddl_schema}}`: The DDL from Stage 1, supplying necessary schema context
– `{{dialect}}`: Optional SQL dialect specification (sqlite, postgresql, etc.) for correct interpretation of dialect-specific syntax

The assembled prompt specifies a strictly structured response format: the model must return a single JSON object with exactly two keys, "verdict" and "explanation" (Figure 7). The "verdict" field is constrained to one of four categorical labels:

– **Correct:** the query fully and accurately answers the question with no semantic issues.
– **Partially correct:** the query is mostly correct but exhibits minor deficiencies, such as missing DISTINCT clauses when duplicates are possible, imprecise date/time boundaries, NULL-handling edge cases, or non-deterministic GROUP BY selections that do not fundamentally invalidate the result.
– **Incorrect:** the query contains fundamental semantic errors, such as wrong table or column selections, missing or incorrect JOIN conditions, flawed aggregation logic, or erroneous WHERE clause predicates.
– **Unanswerable:** the question cannot be answered from the provided schema, due to missing tables or columns, requirements for external data not present in the database, or inherent ambiguity or contradictions in the question itself.

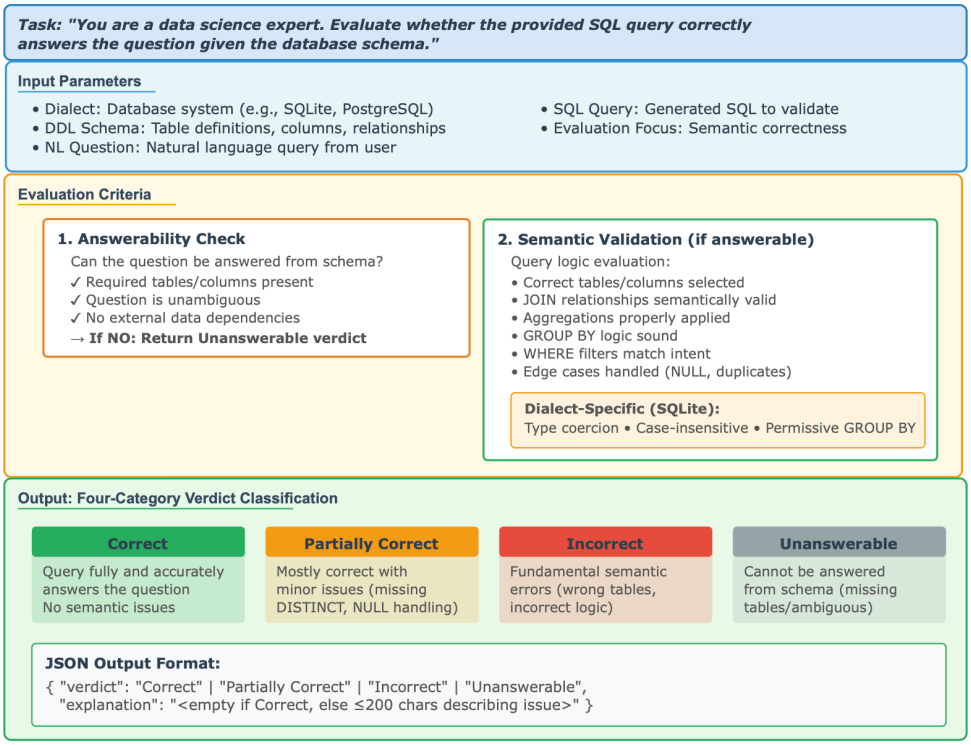

Figure 7. LLM prompt structure with input parameters, evaluation criteria, and structured response format with four-category verdict classification

The "explanation" field encodes a short natural-language justification. For Correct verdicts, the explanation is required to be an empty string, thereby preventing spurious narrative output, while for all other verdicts, the explanation must be a concise single-line description (limited to 160–200 characters) that identifies the dominant semantic issue. Together, these constraints make the LLM judgments machine-parseable and suitable for systematic aggregation across large datasets.

Multi-model consensus voting. This stage queries multiple language models in parallel and collects individual judgments. Through a unified provider interface, the framework supports four model categories: OpenAI, Anthropic, and Google as frontier commercial providers, and Ollama for locally hosted models that run without API costs or rate limits. Combining cross-provider models may mitigate model-specific biases, although residual biases remain [40]. Inter-model agreement serves as a diagnostic indicator for downstream analysis.

Consensus decision. The fourth stage aggregates the individual model responses to determine consensus verdicts and identify cases requiring human review. The categorical verdicts reported in this paper use unweighted agreement. For each of the four categories, the analyzer counts how many models produced that verdict. A majority consensus exists when a single category accounts for more than half of all non-failed model responses, and a verdict is unanimous when all such responses coincide. In our two-model experiment, GPT-5 and Gemini-2.5-pro have equal voting weight. Records lacking a strict majority or receiving strongly divergent model verdicts warrant closer review during dataset curation. They often involve underspecified questions, subtle semantic flaws in SQL queries, or multiple incompatible interpretations.

The implementation also exposes an auxiliary scalar semantic quality score in the closed interval $[0, 1]$ by mapping Correct to 1.0, Partially Correct to 0.5, and both Incorrect and Unanswerable to 0.0, and then computing a configurable provider-weighted average according to formula:

$$\text{score}_{\text{consensus}} = \frac{\sum_{i=1}^{n} w_i \cdot v_i}{\sum_{i=1}^{n} w_i}$$

where v_i denotes the mapped verdict value and w_i the corresponding model weight. The weighted score is useful for deployment settings where providers have known reliability or cost profiles, but it is not used as an authoritative label in our empirical claims.

4. Experimental setup and results

This section reports empirical results from applying the validation framework (Section 3) to all 11,840 examples in Spider 1.0 [7] across three partitions: Test (2,147 examples, 40 databases), Dev (1,034 examples, 20 databases), and Train (8,659 examples, 146 databases). The Train partition comprises both the core 7,000 examples (`train_spider.json`) and the supplementary 1,659 examples (`train_others.json`) distributed with Spider 1.0. The commonly cited total of 10,181 refers to the core splits only (7,000 + 1,034 + 2,147). We audit the full release to ensure completeness. The dataset was obtained in January 2025 from the official Spider resources maintained by the Yale LILY group [4]. The experimental infrastructure employed the technology stack described in Section 3, with SQLite 3.45.0 as the target dialect consistent with Spider's original database format [7, 41]. LLM-based semantic validation used OpenAI GPT-5 [42] with default reasoning effort level of "medium" and Google Gemini-2.5-pro [43] with default thinking budget of 8,192 tokens (both accessed November 2025). These fixed reasoning configurations ensure consistency across all experimental partitions.

The analysis pipeline operated in the streaming architecture described in Section 3, with analyzer components configured per the methodology detailed in Sections 3.3–3.6. Schema validation executed once per unique database identifier (206 total) with cached metadata reused for queries sharing databases. Query-level analyzers operated independently on each SQL statement. Antipattern detection followed the SQLite-specific severity configuration (Section 3.5) with 13 antipatterns classified across four severity levels (Critical, High, Medium, Low). Execution testing used the query execution analyzer configured with `mode=all` (see Section 3), which executes SELECT statements with automatic LIMIT 1 injection when no LIMIT clause is present, and tests UPDATE/DELETE/INSERT statements inside a rollback transaction while blocking destructive operations (e.g., DROP/TRUNCATE/ALTER). Semantic judging used two-model consensus with verdict categories (Correct/Partially Correct/Incorrect/Unanswerable) as defined in Section 3.6. Configuration parameters included `schema_mode=full`, `num_examples=2`, and parallel voting enabled (`parallel_voters=true`, `max_workers=2`).

Results are organized across five quality dimensions reported in Sections 4.1–4.5: (i) database schema integrity (Section 4.1); (ii) SQL syntactic structure (Section 4.2);

(iii) dynamic executability (Section 4.3); (iv) antipattern prevalence (Section 4.4); and (v) LLM-based assessment of NL–SQL semantic correspondence (Section 4.5). Sections 4.6–4.7 report computational performance and integrated quality assessment.

4.1. Database schema integrity assessment

Table 3 presents comparative schema validation results across all three Spider 1.0 partitions.

Table 3. Comparative database schema validation results across Spider 1.0 partitions

Metric	Spider test	Spider dev	Spider train
Total databases	40	20	146
Databases without errors	37 (92.5%)	16 (80.0%)	131 (89.7%)
Databases with errors	3 (7.5%)	4 (20.0%)	15 (10.3%)
Total tables	180	80	793
Empty tables	0 (0%)	0 (0%)	71 (9.0%)
Total foreign keys	146	64	717
Structurally invalid foreign keys	3	2	46
Databases with FK data violations	1	3	6

The results reveal a clear quality gradient across partitions, with Test achieving a 92.5% error-free schema rate, Train 89.7%, and Dev 80.0%. This ordering contradicts the intuitive expectation that training data should exhibit higher quality than test data.

Foreign key structural errors total 51 across all partitions (3 in Test, 2 in Dev, and 46 in Train). Table 4 presents the distribution of schema-integrity findings across partitions.

Table 4. Schema-integrity findings by type and partition

Finding type	Spider test	Spider dev	Spider train	Total	Severity
Structural errors					
FK missing table	0	0	0	0	Critical
FK missing column	3	0	5	8	Critical
FK arity mismatch	0	0	0	0	Critical
FK target not key	0	2	41	43	High
Duplicate columns	0	0	0	0	Medium
Subtotal (structural)	3	2	46	51	—
Data violations					
FK data violations (rows)	1	2,403	39,523	41,927	Critical
Databases with data violations	1	3	6	10	—

The 51 structural foreign-key errors consist of 43 *Target Not Key* and eight *Missing Parent Column* cases. Referential integrity violations total 41,927 rows across 10 databases.

Detailed per-database breakdowns of data violations and foreign-key findings are provided in Appendix A (Tables A1–A3). The violations are highly concentrated. Three databases account for 99.97% of all referential integrity violations: sakila_1 in Train (38,273 rows, 91.3%), flight_2 in Dev (2,400 rows, 5.7%), and flight_4 in Train (1,240 rows, 3.0%). The remaining seven affected databases each contribute fewer than ten violations.

The Train partition also contains 71 empty tables across 9 databases (Table A3), seven of which have 0% data coverage. Execution-based checks can be less informative when queries reference empty tables, because SQL that does not match the intended question may still return empty result sets or zero-valued aggregates.

4.2. Syntactic structure and complexity analysis

All 11,840 queries across all three partitions were successfully parsed by SQLGlot, enabling AST-based feature extraction and difficulty classification for the full corpus.

Examination of difficulty level distributions (Table 5) reveals nearly identical proportions across Test and Dev: approximately 24% Easy, 58–59% Medium, and 17% Hard. Train contains 66 Expert-level queries (0.8%), while Test and Dev contain none.

Table 5. Query difficulty distribution across partitions

Difficulty level	Spider test	Spider dev	Spider train
Easy	516 (24.0%)	250 (24.2%)	2,039 (23.5%)
Medium	1,246 (58.0%)	612 (59.2%)	4,827 (55.7%)
Hard	385 (17.9%)	172 (16.6%)	1,727 (19.9%)
Expert	0 (0.0%)	0 (0.0%)	66 (0.8%)

Analysis of JOIN distribution (Table 6) reveals a second dimension where Train exhibits systematically higher complexity compared to Test and Dev. Single-table queries dominate all partitions (56–61%). Train exhibits higher multi-table query representation (44.02%) compared to Test (40.15%) and Dev (39.46%), with particularly elevated 3–JOIN and 4+–JOIN frequencies.

Table 6. JOIN distribution analysis

JOIN count	Spider test	Spider dev	Spider train
0 (single-table)	1,285 (59.85%)	626 (60.54%)	4,847 (55.98%)
1 JOIN	558 (25.99%)	320 (30.95%)	2,233 (25.79%)
2 JOINs	242 (11.27%)	72 (6.96%)	1,042 (12.03%)
3 JOINs	38 (1.77%)	10 (0.97%)	359 (4.15%)
4+ JOINs	24 (1.12%)	6 (0.58%)	178 (2.06%)
Multi-table (≥1 JOIN)	862 (40.15%)	408 (39.46%)	3,812 (44.02%)

Despite the rich structural variation in JOIN complexity and overall difficulty levels, Spider 1.0 exhibits complete absence of certain advanced SQL features. No queries in any partition employ common table expressions (WITH clauses), window functions (ROW_-NUMBER, RANK, LAG), or recursive queries.

4.3. Dynamic executability validation

Table 7 presents query execution results.

Table 7. Query execution results

Metric	Spider test	Spider dev	Spider train
Total queries	2,147	1,034	8,659
Successfully executed	2,147 (100%)	1,034 (100%)	8,656 (99.97%)
Execution failures	0 (0%)	0 (0%)	3 (0.03%)
Failed items	—	—	3154, 4514, 4515
Mean execution time (ms)	3.10	2.49	1.19

Test and Dev reach 100% execution success. Train reaches 99.97%, failing on 3 of 8,659 queries. All three stem from defective gold SQL. Item 3154 (database `assets_maintenance`)

joins `Ref_Company_Types`, a table absent from its database. SQLite therefore returns "no such table". Items 4514 and 4515 (database `document_management`) place `ORDER BY` and `LIMIT` inside the first `SELECT` of an `INTERSECT`. SQLite permits these clauses only once, after the final `SELECT`, and rejects the query at prepare time. All three are included in the manual validation reported in Section 4.5.

Mean execution time stays below 3.5 ms in every partition. Test (3.10 ms) and Dev (2.49 ms) run slower per query than Train (1.19 ms). Their queries tend to scan more rows per statement, which raises I/O and execution cost.

4.4. Code quality assessment via antipattern detection

Table 8 presents overall code quality metrics.

Table 8. Overall code quality metrics

Metric	Spider test	Spider dev	Spider train
Queries without antipatterns	1,774 (82.6%)	865 (83.7%)	7,496 (86.6%)
Total antipattern occurrences	381	169	1,180

Most queries contain no detected antipatterns: 82.6% (Test), 83.7% (Dev), 86.6% (Train).

Table 9 presents antipattern distribution by type and severity.

Table 9. Antipattern distribution by severity

Antipattern	Spider test	Spider dev	Spider train	Severity
Cartesian product	2 (0.1%)	2 (0.2%)	13 (0.2%)	Critical
Missing GROUP BY	239 (11.1%)	106 (10.3%)	600 (6.9%)	High
NOT IN with nullable	74 (3.4%)	46 (4.4%)	228 (2.6%)	High
Leading wildcard LIKE	50 (2.3%)	12 (1.2%)	140 (1.6%)	Medium
Function in WHERE	0 (0.0%)	0 (0.0%)	6 (0.1%)	Medium
Redundant DISTINCT	4 (0.2%)	0 (0.0%)	131 (1.5%)	Low
SELECT *	12 (0.6%)	3 (0.3%)	62 (0.7%)	Low

Missing GROUP BY is the most frequent high-severity antipattern, with 945 detections overall (6.9–11.1% of queries per partition). NOT IN with nullable columns affects 2.6–4.4% of queries. All 17 critical Cartesian products (2 Test, 2 Dev, 13 Train) are confirmed as genuine join bugs in our manual calibration audit (Section 4.5).

The Missing GROUP BY prevalence is significant because it may reflect SQLite's permissive handling of aggregates with non-aggregated "bare" columns. The SQLite documentation characterizes this behavior as an intentional extension [37]. Under the SQL standard [36], non-aggregated SELECT columns must be grouped unless functionally dependent on grouped columns. PostgreSQL and MySQL (with `ONLY_FULL_GROUP_BY` enabled) reject queries lacking this dependency [38, 44]. Models trained on such examples may therefore learn dialect-specific SQL that fails or returns non-deterministic results. The hazard is not only cross-engine. Even in SQLite, a bare column is taken from an arbitrary row of its group when the grouping key does not determine it. The same gold query can then return different rows for different physical orderings. The two Spider Train cases below illustrate this. Item 1580/1581 asks for counts per account, but the SQL has no GROUP BY and returns one global count with an arbitrary `account_id`. Item 4522/4523 asks for

all users with the most common role, but the SQL groups only by `role_code` and projects `user_name`/`password` as bare columns. SQLite selects one arbitrary representative row for that role. Then `ORDER BY count(*) DESC LIMIT 1` keeps only that one row instead of all matching users. The semantic committee independently identified the same bare-column defect and judged both cases unanimously Incorrect (Section 4.5).

Missing GROUP BY: non-deterministic bare columns (Spider Train)

Item 1580/1581: *Question: Count transactions per account.*

```
SELECT count(*), account_id FROM Financial_transactions
```

Item 4522/4523: *Question: List all users with the most common role.*

```
SELECT user_name, password FROM users GROUP BY role_code
ORDER BY count(*) DESC LIMIT 1
```

Similarly, the 348 queries using NOT IN with potentially nullable columns introduce a subtle correctness risk rooted in SQL's three-valued logic [36]. If the subquery returns any NULL value, NOT IN evaluates to FALSE for matches and UNKNOWN otherwise. The WHERE clause then silently filters out all rows and returns an empty result set [23, 34].

Cross-benchmark validation and baseline comparison. In a separate companion study by the same authors [45], we evaluated the antipattern detector on three widely used Text-to-SQL benchmarks: Spider 1.0, BIRD, and the synthetic Gretel corpus, which together contain 122,802 queries. That evaluation uses the same detection rules as this paper, with one exception: it applies a broader cross-join rule that additionally flags implicit joins whose linking predicate is placed in the `WHERE` clause rather than in `ON`. This paper reports the more conservative Cartesian-product count, which is why the per-class Cartesian totals differ slightly between the two studies. This evaluation serves two purposes. First, it shows that the detector generalizes beyond Spider 1.0: the same structural defects appear in all three benchmarks, although their distribution differs from one to another. The synthetic Gretel corpus stands out, with more severe defects such as destructive DML statements and syntactically invalid SQL. Second, it compares the detector against SqlCheck [24] as a SQL-only baseline. On a stratified sample of 1,237 queries, the proposed detector covered all antipattern classes, whereas SqlCheck reproduced the expected verdict in only 58.6% of cases and covered four classes. The gap was largest for Cartesian products: SqlCheck reached 29.5% recall and missed eight distinct join-defect patterns, including comma joins and tautological `ON 1=1` predicates.

4.5. LLM-based semantic judgments

Table 10 presents semantic judgments from the two-model LLM committee.

Table 10. Semantic judgments from the two-model LLM committee

Consensus verdict	Spider test	Spider dev	Spider train
Correct	1,409 (65.6%)	684 (66.2%)	5,188 (59.9%)
Partially correct	43 (2.0%)	14 (1.4%)	259 (3.0%)
Incorrect	171 (8.0%)	94 (9.1%)	867 (10.0%)
Mixed (disagreement)	522 (24.3%)	242 (23.4%)	2,316 (26.7%)
Unanswerable	2 (0.1%)	0 (0.0%)	29 (0.3%)
Total	2,147 (100%)	1,034 (100%)	8,659 (100%)

The proportion of items assigned a Correct judgment is similar in Test and Dev (65.6% and 66.2%) and lower in Train (59.9%). Across all partitions, 1,132 items receive an Incorrect judgment (8.0–10.0% per partition). Model disagreement (Mixed) affects 23.4–26.7% of queries across partitions.

A detailed breakdown of the Mixed category by disagreement direction is provided in Appendix A (Table A4). The dominant pattern is the Correct ↔ Partially Correct boundary (∼60–62% of Mixed cases across all partitions), consistent with instances where the SQL largely matches intent but differs in minor constraints (e.g., DISTINCT/NULL handling). The next-largest group is Correct ↔ Incorrect (∼25–29%), where judges differ in how strictly they interpret semantic mismatches. Disagreement patterns involving an Incorrect judgment therefore remain high-priority candidates for manual review.

Inter-model agreement. To formally characterize the extent of these disagreements, Table 11 quantifies agreement between Gemini-2.5-pro and GPT-5 using Cohen's κ [46]. The qualitative labels follow the Landis–Koch interpretation bands [47]. On the full four-category scale the two judges achieve $\kappa = 0.42$ (moderate agreement) with 74.0% raw agreement over all 11,840 items. When categories are collapsed to a binary Accept (Correct ∪ Partially Correct) versus Reject (Incorrect ∪ Unanswerable) distinction, agreement rises to $\kappa = 0.61$ (substantial), with 90.0% raw agreement. This pattern is consistent with the disagreement analysis. Four-category agreement is moderate, whereas binary agreement is substantial. The dominant source of discord is the Correct ↔ Partially Correct boundary (Table A4), where judges differ on whether minor issues constitute a partial defect or are negligible.

Table 11. Inter-model agreement between Gemini-2.5-pro and GPT-5

Partition	N	Raw agr. (4-cat)	κ (4-cat)	Raw agr. (binary)	κ (binary)
Spider test	2,147	75.7%	0.39 (fair)	90.7%	0.59 (moderate)
Spider dev	1,034	76.6%	0.41 (moderate)	90.9%	0.62 (substantial)
Spider train	8,659	73.3%	0.43 (moderate)	89.7%	0.62 (substantial)
Total	11,840	74.0%	0.42 (moderate)	90.0%	0.61 (substantial)

Manual validation. We calibrate the two-model consensus against author judgment. We manually re-execute and adjudicate 367 unique flagged items across seven audit groups, verifying each automated verdict against execution evidence rather than treating it as ground truth. Each gold query is run against the actual database and compared with a hand-written corrected control. This separates two axes that a single label conflates. The first is execution-correctness: does the query return the wrong rows on *this* data? The second is robustness: would it break on *other* data? We report the per-subset outcomes below.

(1) Cartesian products (all 17). Both the Cartesian-product detector and the LLM committee flag these items. Every one is a genuine bug, since the cross join materializes a wrong result set. Nine are missing-`ON` joins, one of which returns all 18,228 players instead of 13. The other eight are degenerate self-referential predicates where the join condition compares a column with itself.

(2) Consensus Unanswerable (all 31). Twenty-six items are confirmed non-answerable from the provided schema. Two causes account for them. The first is a *missing concept:* the question asks for data the schema does not store. "Population density" needs a city-area column. A "best paper award", a publication "month", a "flagship" store type, and a product "category" have no corresponding field. The second is a *structural constraint*

that makes the question ill-posed. “Through how many states does the river run” cannot be counted because `river.traverse` stores a single state. “Customers with both a saving and a checking account” is impossible because the schema allows one account per customer. The remaining five items, all in `soccer_2`, are false positives caused by schema serialization. Our prompt exposes the database DDL but omits Spider 1.0’s `tables.json`, which defines `HS` as *training hours*. Without this metadata, both judges labelled these answerable questions unanswerable. This is the only audited case in which omitted metadata changes the verdict. We therefore exclude these items from the final list.

(3) Unanswerable–Incorrect splits (all 47). Independent review classifies 34 items as Unanswerable, 8 as Partially Correct, and 5 as Incorrect. Paraphrase drift is a major source of errors. For at least 21 of these items, another Spider 1.0 question paired with the same gold SQL states the intended condition more clearly. Common changes include dropping an explicit year, a comparison criterion, or an entity. Other Unanswerable cases rely on undocumented terms such as “major,” “top,” or “best.” Some lack a required concept or relationship, have a structural schema limitation, or contain a truncated question. The Partially Correct cases have a defensible benchmark reading but remain ambiguous or fragile. These include the undeclared `movie.mid = *.msid` convention used throughout the `imdb` subset. The five Incorrect items ask answerable questions, but the gold SQL uses the wrong table, relationship, aggregation, or projection. Examples include using assessment dates instead of course-attendance dates, collapsing all movies by a director into one row, and returning `lowest_point` instead of the requested elevation.

(4) Incorrect–Incorrect (random, 212). Manual validation confirms defects in 130 of 212 items (about 61%). The confirmed cases include incorrect WHERE clauses, aggregation and relationship errors, duplicate over-counts, and three `flight_2` items affected by database-level whitespace errors. In a few cases the flawed logic coincidentally returns the correct rows on this database, yet the query is semantically wrong and would fail on other data. Another 59 items (28%) are Partially Correct. Their issues include missing `DISTINCT` clauses, the use of `INNER` instead of `LEFT` joins, lexicographic sorting of `TEXT` values, and the non-deterministic SQLite bare-column `min()`/`max()` pattern discussed earlier. A further 16 items (8%) reflect natural-language ambiguity, such as whether “greater than any” means greater than the minimum or the maximum. The remaining 7 items (3%) are false positives caused by benchmark-specific conventions or paraphrase-consistent readings not captured by either judge.

(5) Execution failures (all 3). The three queries that failed dynamic execution in Section 4.3 were manually confirmed as defective, and both judges classified each as Incorrect.

(6) Consensus Partially Correct (random, 30). Twenty-eight items are problematic: 25 return duplicate rows because `DISTINCT` is missing, 2 return incorrect counts because they count rows instead of distinct entities, and 1 has a latent string-matching defect. The other 2 are false positives because repeated values correspond to distinct requested entities.

(7) Partially Correct–Incorrect (random, 30). Nineteen items are Incorrect: 10 have aggregation or counting errors, 4 omit required rows or columns, 2 sort a `TEXT` column lexicographically rather than numerically, 1 changes the result granularity, 1 uses an incorrect NULL predicate, and 1 collapses rows with an over-eager `DISTINCT`. Four items are Partially Correct: one returns the requested values plus an extra column, and three return duplicate rows. The remaining 7 are false positives.

In total, manual validation confirms genuine defects in 263 unique Spider 1.0 items, listed in Table 12 under six groups: 17 Cartesian-product bugs, 26 consensus Unanswer-

Table 12. Manually confirmed problematic Spider 1.0 items by audit group and partition (266 entries; 263 unique items)

Audit group	Part.	n	Spider item IDs
Cartesian product	Dev	2	945, 946
	Test	2	388, 389
	Train	13	3132, 3662, 3663, 5779, 5780, 6796, 6797, 6798, 6799, 6800, 6801, 6802, 6803
Consensus unanswerable	Test	2	1129, 1265
	Train	24	155, 1017, 2534, 3037, 3038, 3394, 4393, 4926, 5173, 5174, 5352, 6582, 6583, 7295, 7299, 7301, 7394, 7399, 7487, 7488, 7489, 8022, 8024, 8025
Unanswerable ∧ Incorrect	Dev	2	176, 643
	Test	2	1156, 1710
	Train	35	76, 77, 2197, 2418, 2504, 2799, 2838, 2841, 4329, 4335, 4385, 5009, 5197, 5241, 6220, 7178, 7257, 7258, 7260, 7279, 7281, 7296, 7352, 7417, 7483, 7540, 7707, 7720, 7779, 7834, 7840, 7857, 7861, 8087, 8100
Incorrect ∧ Incorrect	Dev	14	97, 103, 167, 223, 229, 252, 501, 527, 642, 885, 908, 917, 943, 946
	Test	32	142, 388, 442, 443, 465, 673, 698, 708, 727, 728, 758, 927, 993, 1080, 1112, 1173, 1245, 1321, 1425, 1448, 1469, 1551, 1575, 1677, 1840, 1844, 1878, 1901, 1994, 2004, 2114, 2128
	Train	84	59, 89, 90, 156, 273, 278, 316, 511, 565, 567, 625, 626, 641, 682, 719, 870, 873, 900, 925, 1223, 1306, 1455, 1519, 1622, 1623, 1687, 2038, 2174, 2199, 2225, 2357, 2358, 2364, 2379, 2380, 2407, 2420, 2461, 2517, 2520, 2579, 2584, 2732, 2852, 2857, 3140, 3252, 3364, 3425, 3560, 3588, 3589, 3638, 3663, 3672, 3686, 3691, 3694, 3695, 3710, 3802, 3900, 3901, 3949, 3965, 3972, 4061, 4072, 4327, 4334, 4392, 4443, 4444, 4456, 4462, 4523, 4546, 4962, 5424, 5433, 5535, 5590, 5740, 5811
Execution failures	Train	3	3154, 4514, 4515
Partially correct–Incorrect + Consensus partially correct	Dev	4	67, 370, 371, 531
	Test	13	149, 548, 761, 1041, 1091, 1101, 1152, 1179, 1412, 1514, 1621, 2010, 2081
	Train	34	65, 66, 334, 335, 513, 1320, 1461, 2093, 2218, 2253, 2257, 2258, 2511, 2512, 2645, 2646, 3315, 3320, 3435, 3784, 4227, 4258, 4965, 4966, 5118, 5128, 5145, 5488, 5767, 5859, 6355, 6532, 6991, 6992
Total		266	

able questions, 39 Unanswerable–Incorrect splits (34 Unanswerable + 5 Incorrect), 130 Incorrect–Incorrect items, 3 execution failures, and 51 items from the combined Partially Correct audit strata. Machine-readable copies of these items are released in the repository under the MainSpiderResults/ManualValidation folder. Each line is a JSON object with the question, gold SQL, committee verdicts, and our adjudication.

Three of the seven audited groups were random samples, so we report 95% Wilson score confidence intervals for their stratum-specific confirmation rates [48]. The confirmed rate is 61.3% for the random Incorrect–Incorrect sample (130 of 212, CI 54.6–67.6%), 93.3% for consensus Partially Correct (28 of 30, CI 78.7–98.2%), and 76.7% for Partially Correct–Incorrect (23 of 30, CI 59.1–88.2%). The other four groups were audited exhaustively, so no sampling interval applies.

External validation. To further corroborate our findings, we compared the Dev-set items flagged as semantically problematic in [20] against the outcomes of our framework. For each of the sixteen items we executed the flagged query against the Spider 1.0 database, inspected the result set, and verified manually whether the query correctly answers the natural language question. The sixteen items correspond to eleven distinct queries, since five recur under paraphrased questions. Table 13 reports the per-query outcomes.

Table 13. Cross-check of sixteen Dev-set items marked invalid in [20] against our framework and manual review. Paired indices share byte-identical SQL under paraphrased questions.

Dev #	Our framework	Manual	Failure note
67	Mixed	Partially Correct	Redundant projected column `age`
101	Correct	Correct	Correct; "name" denotes `Maker` per gold
128, 129	Correct	Correct	Semantically and logically correct
177	Mixed	Partially Correct	`COUNT(*)` vs `COUNT(DISTINCT)` may overcount on non-unique model names
494	Incorrect	Incorrect	Omitted column `result`
555	Mixed	Incorrect	Case-sensitive literal; empty result
579	Incorrect	Incorrect	Returns a count, not the list
773, 774	Incorrect	Incorrect	`MIN` where the question implies `MAX`
811, 812	Correct	Incorrect	Case-sensitive literal; `NULL` aggregate result
819, 820	Incorrect	Incorrect	Grouped count, not a single scalar total
961, 962	Incorrect	Incorrect	`MAX` where the question requests the youngest; lexicographic `age`

Three items that [20] marks invalid are in fact correct: item 101 and the paired 128 and 129. Each executes successfully and reproduces the gold result. For item 101, Spider uses "full name" for `FullName`, so "name" is interpreted as `Maker`. Our framework also produced false negatives. Gemini-2.5-pro and GPT-5 failed to flag 555, 811, and 812 as incorrect, even though all three return an empty or `NULL` result due to a case-sensitivity mismatch. Item 177 is partially correct, returning the expected result on this instance while relying on `COUNT(*)` over a text join.

4.6. Computational performance characteristics

Experiments were executed on a single workstation (macOS 14.x, Apple M1 Pro, 16 GB RAM). Table 14 presents processing time by analyzer component.

Formal analyzers (antipattern, syntax, execution, schema) complete within 20 milliseconds per query. The Semantic LLM Judge requires 11.7–13.7 seconds per query and accounts for 99.9% of total runtime. Cumulative analysis time for all 11,840 examples totals approximately 43.6 hours.

4.7. Integrated quality assessment and recommendations

Table 15 synthesizes key results across all validation dimensions for comparative assessment.

Table 14. Processing time by analyzer component and partition

Component	Spider test	Spider dev	Spider train
Per-query latency (ms/query)			
Antipattern detection	1.21	0.49	0.56
Syntax analysis	2.32	0.60	0.64
Execution testing	3.10	2.49	1.19
Schema validation	14.25	16.91	9.93
Semantic LLM judge	11,707	13,005	13,682
Aggregate wall-clock time			
Fast analyzers + I/O	∼1 min	∼30 sec	∼1.3 min
LLM judge	∼7.0 hr	∼3.7 hr	∼32.9 hr
Total pipeline	∼7.0 hr	∼3.7 hr	∼32.9 hr

Table 15. Integrated quality assessment across all validation dimensions

Quality dimension	Spider test	Spider dev	Spider tgrain
Database schemas			
Schemas without errors (%)	92.5	80.0	89.7
Structural FK errors (count)	3	2	46
FK data violations (rows)	1	2,403	39,523
Syntactic quality			
SQLGlot parse success (%)	100	100	100
Medium + difficulty (%)	76.0	75.8	76.5
Executability			
Execution success (%)	100	100	99.97
Code quality			
Critical antipatterns (count)	2	2	13
High-severity antipatterns (%)	14.5	14.7	9.6
LLM-based semantic judgments			
Correct (%)	65.6	66.2	59.9
Incorrect (%)	8.0	9.1	10.0
Disputed/mixed (%)	24.3	23.4	26.7

The integrated assessment reveals consistent patterns across all partitions. Parsing and execution are perfect or near-perfect, and 82.6–86.6% of queries are free of detected antipatterns. However, 7.5–20.0% of databases contain foreign-key structural errors, row-level referential-integrity violations, or both. At the item level, 59.9–66.2% of examples receive a Correct consensus verdict. Dev exhibits the lowest error-free schema rate (80.0%) despite its critical role in model selection.

Taken together, the results define four groups of remediation targets. First, database-level repairs should address critical integrity defects and empty-schema databases, summarized in Appendix A (Table A5). Second, query-level repairs should prioritize the confirmed problematic items enumerated in Table 12. Third, independently reported semantic issues should be reconciled using consistent interpretation criteria, as illustrated in Table 13. Fourth, portability-oriented cleanup should annotate or repair high-severity antipatterns such as missing `GROUP BY` and nullable `NOT IN`, even when they do not always materialize as wrong answers on the current SQLite data.

5. Discussion

This section interprets the empirical findings reported in Section 4. The findings indicate that reliable Text-to-SQL benchmark curation requires multi-dimensional validation, documented interpretation rules, and execution-grounded adjudication of flagged items.

5.1. Hidden defects and evaluation bias

Spider 1.0 appears highly reliable under traditional surface checks: all queries parse successfully and 99.97–100% execute. However, this formal correctness hides structural defects that standard Text-to-SQL evaluation does not capture. The 51 structural foreign-key errors and 41,927 referential-integrity violations remain mostly invisible because SQLite does not enforce foreign-key constraints unless explicitly enabled [41]. The risk is amplified by partition heterogeneity: the error-free schema rate is highest in Test (92.5%), lower in Train (89.7%), and lowest in Dev (80.0%), even though Dev is commonly used for hyperparameter tuning, prompt selection, and model selection [3, 7].

In the `flight_2` database, leading spaces in every `FLIGHTS.SourceAirport` and `FLIGHTS.DestAirport` value prevent both foreign-key joins from matching any of the 1,200 flights, producing 2,400 referential-integrity violations. This database is used by 80 Dev examples (7.7% of the split), and the whitespace defect causes the gold queries for 50 of them to return an empty or incorrect result. Per-item results are provided in `flight_2_execution_audit_80.jsonl` under `MainSpiderResults/SpiderDevDataset_For_Article`.

Execution accuracy compares the outputs of predicted and gold SQL, so a prediction that accounts for the leading spaces may return the intended rows but still be scored as incorrect. Conversely, a prediction that does not compensate for the padding may match the gold output and be scored as correct. For these items, the metric may therefore favor agreement with a faulty reference result over semantic correctness.

5.2. Antipatterns as quality signals

Antipattern detection provides deterministic, inexpensive, and interpretable analysis across the full corpus. The strongest evidence comes from the Cartesian-product class: all 17 critical cases identified by the detector were judged unanimously Incorrect by the semantic committee and confirmed by manual execution.

Other antipattern classes are better treated as robustness or portability signals than as proof of an incorrect answer on the current SQLite data. Some of the 945 missing-`GROUP BY` cases rely on SQLite's bare-column behavior and may fail or become non-deterministic under stricter engines. The 348 nullable-`NOT IN` cases risk three-valued-logic failures when NULL values occur. Manual rechecks confirmed that some flagged patterns produce wrong answers, whereas others are currently correct but brittle. Antipattern detection thus complements LLM judging by revealing both actual SQL defects and broader robustness or portability risks.

5.3. Semantic judging requires manual calibration

LLM judging enables semantic triage across the full corpus. In our audit, consensus Incorrect and Unanswerable verdicts identify high-priority candidates for review. Mixed verdicts identify cases requiring closer examination because they may reflect ambiguity, subtle

semantic defects, or differences in judging strictness. However, an LLM verdict should not be treated as a final label.

The audit shows why this calibration is necessary. Each audited query was re-executed against the actual database and compared with an author-written control query. This separates execution-correctness from robustness: some queries return wrong rows on the current data, while others return the expected result here but remain fragile under realistic data changes. In total, 263 unique items are confirmed as problematic, including 26 consensus Unanswerable items and 17 critical Cartesian-product cases. At the same time, some disputed or unanimously flagged items are better classified as fragile, ambiguous, or dependent on benchmark conventions rather than simply wrong.

Many disputed cases are dataset-design problems rather than simple annotation mistakes. Some questions require concepts absent from the schema, such as ownership, flagship status, product category, or publication month. Other issues include correctable typos, truncated questions, hard-coded temporal anchors, benchmark-specific conventions, and ambiguous quantifiers such as "any". Our cross-check against prior semantic-issue reports [20] reinforces this point: some items marked invalid in earlier analyses are correct under defensible alternative criteria, while execution-level evidence revealed semantic errors missed by our framework.

5.4. Threats to validity

LLM-based semantic validation can introduce systematic biases. First, benchmark contamination is possible: frontier models may have encountered Spider 1.0 or related artifacts during training. Because their training corpora are not fully disclosed, this threat cannot be ruled out.

Second, two-model consensus reduces reliance on any single judge but does not eliminate correlated errors. Both models may share failure modes on SQL semantics, including NULL behavior, duplicate elimination, boundary conditions, dialect-specific aggregate behavior, or domain-specific conventions. We mitigate this threat through the execution-grounded adjudication described in Section 4.5. Nevertheless, consensus outcomes remain strong triage signals rather than definitive labels.

Third, the semantic analyzer receives sampled cell values (Section 3.6). Samples may miss edge cases needed for accurate judgment, such as rare categories, extreme values, duplicates, or NULL prevalence. This risk is especially relevant for aggregates, distribution-dependent predicates, and string comparisons.

Finally, the validation framework has been evaluated end-to-end only on Spider 1.0 with SQLite. Cross-benchmark evidence is currently limited to the antipattern detection component, which we evaluated on Spider 1.0, BIRD, and Gretel in a separate companion study by the same authors [45]. Broader multi-benchmark validation of all framework components remains future work.

6. Conclusions and future work

6.1. Summary of contributions

This paper presents `text2sql-dataset-analyzer`, an open-source framework for auditing Text-to-SQL datasets across five quality dimensions: database schema integrity, SQL

syntactic structure and complexity, dynamic executability, rule-based antipattern detection, and LLM-based assessment of NL–SQL semantic correspondence. The framework produces dataset-level summaries and per-item annotations that support triage, curation, and regression tracking.

Applying the framework to all 11,840 Spider 1.0 examples shows that traditional checks substantially overestimate dataset health. Although 99.97–100% of queries pass execution checks, schema analysis uncovers 51 structural foreign-key errors and 41,927 referential-integrity violations. These defects are highly concentrated and unevenly distributed across partitions. Dev has the lowest error-free schema rate (80.0%). In the `flight_2` database, the gold queries for 50 of its 80 Dev examples return an empty or incorrect result.

The two-model LLM committee assigns Correct judgments to 60–66% of items and Incorrect judgments to 8–10%, while 23–27% fall into the Mixed (disagreement) category. Manual validation confirms that the committee is useful for triage: 263 unique items are genuinely problematic, 26 consensus Unanswerable items are non-solvable under the provided schemas, and all 17 critical Cartesian-product joins are real bugs. At the same time, manual rechecks show that final labels require explicit interpretation criteria because some cases are fragile, ambiguous, or dependent on benchmark conventions rather than simply wrong. The audit also surfaces systematic portability and robustness hazards in gold SQL, including missing `GROUP BY` and nullable `NOT IN`.

These findings have practical implications in three areas. Dataset creators can apply the released framework to audit their Text-to-SQL corpora, identify problematic examples, prioritize repairs, and rerun quality checks after updates. For Spider 1.0, the 263 confirmed items in Table 12 give practitioners a ready list to repair or exclude before training or evaluation. Studies evaluating Text-to-SQL models should specify the benchmark version and document how known dataset issues were handled.

6.2. Limitations and future work

Full semantic validation remains expensive. End-to-end judging over 11,840 examples required approximately 43.6 hours and was dominated by API-based LLM calls. Future work should reduce this cost by using lightweight semantic classifiers to prioritize high-risk items for LLM judging.

Our empirical evaluation used two LLM judges. Future work should test whether larger, more diverse committees and stratified human spot checks improve reliability, particularly for disputed cases shaped by differing interpretation criteria and benchmark conventions. Judging context could also be enriched with compact database summaries, including stratified value samples, NULL rates, duplicate indicators, and distinct-count estimates.

The rule-based query difficulty classifier could be augmented with a learned complexity model. For example, bithreshold neural-network regressors [49] could map extracted structural features to numerical complexity scores, enabling data-driven calibration of difficulty thresholds.

Finally, we do not fine-tune or re-evaluate any Text-to-SQL model on a curated version of Spider 1.0. Future work should measure how repairing the flagged items affects model accuracy.

CRediT authorship contribution statement

Volodymyr Sabadosh: Conceptualization, Software, Data curation, Formal analysis, Methodology, Investigation, Visualization, Writing – original draft, Writing – review and editing. Vladyslav Kotsovsky: Supervision, Writing – review and editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data and code availability

The Text-to-SQL dataset analysis tool developed in this study is available as open-source software on GitHub: https://github.com/vsabadosh/text2sql-dataset-analyzer. The repository includes the configuration files and scripts required to reproduce the analysis, along with precomputed metrics and generated reports for the Spider 1.0 dataset. Detailed, partition-specific outputs (Train/Dev/Test), including the exported reports and artifacts referenced in the paper, are provided in the repository root under the MainSpiderResults directory.

Funding

No funds, grants, or other support was received for this work.

Declaration of AI assistance in the writing process

During the preparation of this manuscript, the authors used ChatGPT (OpenAI) as a supportive tool for language refinement and clarity. In addition, the authors used Cursor with an AI coding assistant (Claude Sonnet) to support implementation tasks (e.g., code suggestions and refactoring) in the development of the proposed framework. All generated suggestions were reviewed, tested, and, where applicable, manually revised by the authors. The authors take full responsibility for the content of the manuscript and the correctness of the software and results reported in this paper.

References

[1] J. Fürst, C. Kosten, F. Nooralahzadeh, Y. Zhang, and K. Stockinger, "Evaluating the data model robustness of Text-to-SQL systems based on real user queries," in *Proceedings of the 28th International Conference on Extending Database Technology (EDBT 2025)*, 2025, pp. 158–170. [Online]. https://doi.org/10.48786/edbt.2025.13

[2] L. Shi, Z. Tang, N. Zhang, X. Zhang, and Z. Yang, "A survey on employing large language models for Text-to-SQL tasks," *ACM Computing Surveys*, Vol. 58, No. 2, 2025, pp. 1–37. [Online]. https://doi.org/10.1145/3737873

[3] D. Gao, H. Wang, Y. Li, X. Sun, Y. Qian et al., "Text-to-SQL empowered by large language models: A benchmark evaluation," *Proceedings of the VLDB Endowment*, Vol. 17, No. 5, 2024, pp. 1132–1145. [Online]. https://doi.org/10.14778/3641204.3641221

[4] Yale LILY Group, "Spider 1.0 – leaderboard," 2025, accessed: 2025-11-08. [Online]. https://yale-lily.github.io/spider

[5] V. Shkapenyuk, D. Srivastava, T. Johnson, and P. Ghane, "Automatic metadata extraction for Text-to-SQL," *CoRR*, Vol. abs/2505.19988, 2025. [Online]. https://doi.org/10.48550/arXiv.2505.19988

[6] BIRD Benchmark Team, "BIRD – leaderboard," 2025, accessed: 2025-11-08. [Online]. https://bird-bench.github.io/

[7] T. Yu, R. Zhang, K. Yang, M. Yasunaga, D. Wang et al., "Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and Text-to-SQL task," in *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Brussels, Belgium: Association for Computational Linguistics, 2018, pp. 3911–3921. [Online]. https://doi.org/10.18653/v1/D18-1425

[8] Snowflake Inc., "Using Snowflake Copilot inline," Snowflake Documentation, 2025, accessed: 2025-11-08. [Online]. https://docs.snowflake.com/en/user-guide/snowflake-copilot-inline

[9] Microsoft, "What is an AI/BI Genie space," Microsoft Learn (Azure Databricks Documentation), 2025, accessed: 2025-11-08. [Online]. https://learn.microsoft.com/en-us/azure/databricks/genie/

[10] Microsoft, "Copilot in Fabric in the SQL database workload," Microsoft Learn, 2025, accessed: 2025-11-08. [Online]. https://learn.microsoft.com/en-us/fabric/database/sql/copilot-sql-database

[11] V. Zhong, C. Xiong, and R. Socher, "Seq2SQL: Generating structured queries from natural language using reinforcement learning," *CoRR*, Vol. abs/1709.00103, 2017. [Online]. https://doi.org/10.48550/arXiv.1709.00103

[12] J. Li, B. Hui, G. Qu, J. Yang, B. Li et al., "Can LLM already serve as a database interface? a BIg bench for large-scale database grounded Text-to-SQLs," in *Advances in Neural Information Processing Systems 36 (NeurIPS 2023), Datasets and Benchmarks Track*, 2023, pp. 42 330–42 357. [Online]. https://proceedings.neurips.cc/paper_files/paper/2023/hash/83fc8fab1710363050bbd1d4b8cc0021-Abstract-Datasets_and_Benchmarks.html

[13] Gretel.ai, "Synthetic Text-to-SQL dataset (Gretel-Synth)," Hugging Face Datasets, 2024, accessed: 2025-11-08. [Online]. https://huggingface.co/datasets/gretelai/synthetic_text_to_sql

[14] H. Li, S. Wu, X. Zhang, X. Huang, J. Zhang et al., "OmniSQL: Synthesizing high-quality Text-to-SQL data at scale," *Proceedings of the VLDB Endowment*, Vol. 18, No. 11, 2025, pp. 4695–4709. [Online]. https://doi.org/10.14778/3749646.3749723

[15] S. Chang, J. Wang, M. Dong, L. Pan, H. Zhu et al., "Dr.Spider: A diagnostic evaluation benchmark towards Text-to-SQL robustness," in *Proceedings of the 11th International Conference on Learning Representations (ICLR)*, 2023. [Online]. https://openreview.net/forum?id=Wc5bmZZU9cy

[16] Z. Yao, G. Sun, Ł. Borchmann, Z. Shen, M. Deng et al., "Arctic-Text2SQL-R1: Simple rewards, strong reasoning in Text-to-SQL," in *Findings of the Association for Computational Linguistics: ACL 2026*. San Diego, California, United States: Association for Computational Linguistics, 2026, pp. 26 966–26 995. [Online]. https://doi.org/10.18653/v1/2026.findings-acl.1345

[17] T. Jin, Y. Choi, Y. Zhu, and D. Kang, "Pervasive annotation errors break Text-to-SQL benchmarks and leaderboards," *Proceedings of the VLDB Endowment*, Vol. 19, No. 5, 2026, pp. 931–944. [Online]. https://doi.org/10.14778/3796195.3796206

[18] A. Mitsopoulou and G. Koutrika, "Analysis of Text-to-SQL benchmarks: Limitations, challenges and opportunities," in *Proceedings of the 28th International Conference on Extending Database Technology (EDBT 2025)*. OpenProceedings.org, 2025, pp. 199–212. [Online]. https://doi.org/10.48786/edbt.2025.16

[19] P. Pandey, D. Patel, S. Mandvikar, and N. Kota, "Ensuring data accuracy in Text-to-SQL systems: A comprehensive validation framework," *International Journal of Computer Trends*

and Technology, Vol. 72, No. 12, 2024, pp. 17–24. [Online]. https://doi.org/10.14445/22312803/IJCTT-V72I12P103

[20] X. Liu, S. Shen, B. Li, N. Tang, and Y. Luo, "NL2SQL-BUGs: A benchmark for detecting semantic errors in NL2SQL translation," in *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '25)*, 2025, pp. 5662–5673. [Online]. https://doi.org/10.1145/3711896.3737427

[21] L. Zheng, W.L. Chiang, Y. Sheng, S. Zhuang, Z. Wu et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," in *Advances in Neural Information Processing Systems 36 (NeurIPS 2023), Datasets and Benchmarks Track*, 2023.

[22] C.M. Chan, W. Chen, Y. Su, J. Yu, W. Xue et al., "ChatEval: Towards better LLM-based evaluators through multi-agent debate," in *Proceedings of the 12th International Conference on Learning Representations (ICLR)*, 2024.

[23] B. Karwin, *SQL Antipatterns: Avoiding the Pitfalls of Database Programming*. Pragmatic Bookshelf, 2010.

[24] P. Dintyala, A. Narechania, and J. Arulraj, "SQLCheck: Automated detection and diagnosis of SQL anti-patterns," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, 2020, pp. 2331–2345. [Online]. https://doi.org/10.1145/3318464.3389754

[25] Z. Abedjan, L. Golab, and F. Naumann, "Profiling relational data: A survey," *The VLDB Journal*, Vol. 24, No. 4, 2015, pp. 557–581.

[26] M. Memari, S. Link, and G. Dobbie, "SQL data profiling of foreign keys," in *Proceedings of the 34th International Conference on Conceptual Modeling (ER 2015)*, 2015, pp. 229–243.

[27] S. Fatehi, "SchemaCrawler: Free database schema discovery and comprehension tool," 2026, accessed: 2026-06-06. [Online]. https://www.schemacrawler.com/

[28] Python Software Foundation, "Python 3.11 documentation," 2025, accessed: 2025-11-08. [Online]. https://docs.python.org/3.11/

[29] T. Mao, "SQLGlot: Python SQL parser and transpiler," 2025, accessed: 2025-11-08. [Online]. https://github.com/tobymao/sqlglot

[30] M. Bayer et al., "SQLAlchemy 2.0 documentation," 2025, accessed: 2025-11-08. [Online]. https://docs.sqlalchemy.org/en/20/

[31] DuckDB Foundation, "DuckDB documentation," 2025, accessed: 2025-11-08. [Online]. https://duckdb.org/docs/stable/

[32] R. Mogylatov, "Dependency Injector: Dependency injection framework for Python," 2025, accessed: 2025-11-08. [Online]. https://python-dependency-injector.ets-labs.org/

[33] PyYAML Developers, "PyYAML – YAML parser and emitter for Python," 2025, accessed: 2025-11-08. [Online]. https://pyyaml.org/

[34] C.J. Date, *SQL and Relational Theory: How to Write Accurate SQL Code*, 2nd ed. O'Reilly Media, 2012.

[35] R. Ramakrishnan and J. Gehrke, *Database Management Systems*, 3rd ed. McGraw-Hill, 2003.

[36] *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*, ISO/IEC Std. 9075-2:2016, 2016. [Online]. https://www.iso.org/standard/63556.html

[37] SQLite, "SELECT – section 2.5: Bare columns in an aggregate query," 2025, accessed: 2025-11-08. [Online]. https://sqlite.org/lang_select.html#bare_columns_in_an_aggregate_query

[38] PostgreSQL Global Development Group, "The GROUP BY and HAVING clauses," 2024, accessed: 2025-11-08. [Online]. https://www.postgresql.org/docs/current/queries-table-expressions.html#QUERIES-GROUP

[39] A. Molinaro and R. de Graaf, *SQL Cookbook: Query Solutions and Techniques for All SQL Users*, 2nd ed. O'Reilly Media, 2020.

[40] H. Chen and S. Goldfarb-Tarrant, "Safer or luckier? LLMs as safety evaluators are not robust to artifacts," in *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, 2025, pp. 19 750–19 766. [Online]. https://aclanthology.org/2025.acl-long.970/

[41] SQLite, "Foreign key support," 2025, accessed: 2025-11-08. [Online]. https://sqlite.org/foreignkeys.html

[42] OpenAI, "Models documentation (API reference)," 2025, accessed: 2025-11-08. [Online]. https://platform.openai.com/docs/models
[43] Google, "Gemini API documentation," 2025, accessed: 2025-11-08. [Online]. https://ai.google.dev/gemini-api/docs
[44] Oracle Corporation, "MySQL handling of GROUP BY," 2024, accessed: 2025-11-08. [Online]. https://dev.mysql.com/doc/refman/8.0/en/group-by-handling.html
[45] V.M. Sabadosh and V.M. Kotsovsky, "Automated detection and comparative analysis of SQL antipatterns in Text-to-SQL datasets," *Ukrainian Journal of Information Technology*, Vol. 8, No. 1, 2026, pp. 135–143. [Online]. https://doi.org/10.23939/ujit2026.01.135
[46] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, Vol. 20, No. 1, 1960, pp. 37–46.
[47] J.R. Landis and G.G. Koch, "The measurement of observer agreement for categorical data," *Biometrics*, Vol. 33, No. 1, 1977, pp. 159–174.
[48] L.D. Brown, T.T. Cai, and A. DasGupta, "Interval estimation for a binomial proportion," *Statistical Science*, Vol. 16, No. 2, 2001, pp. 101–133. [Online]. https://doi.org/10.1214/ss/1009213286
[49] V.M. Kotsovsky and A. Batyuk, "Towards the design of bithreshold ANN regressor," in *Proceedings of the 19th IEEE International Scientific and Technical Conference on Computer Sciences and Information Technologies (CSIT 2024)*. Lviv, Ukraine: IEEE, 2024, pp. 1–4.

Appendix A. Detailed per-database quality tables

This appendix provides granular per-database tables referenced in Sections 4.1–4.7.

Table A1. Databases with foreign key data violations

Database ID	Partition	FK data violations	% of Total
sakila_1	Train	38,273	91.3%
flight_2	Dev	2,400	5.7%
flight_4	Train	1,240	3.0%
car_1	Dev	2	<0.1%
hr_1	Train	6	<0.1%
college_1	Train	2	<0.1%
wta_1	Dev	1	<0.1%
hospital_1	Train	1	<0.1%
allergy_1	Train	1	<0.1%
pilot_1	Test	1	<0.1%
Total	—	41,927	100%

Table A2. Database identifiers by foreign-key finding type (× denotes finding count per database)

Finding type	Partition	Count	Database IDs
FK missing column	Test	3	book_1 (2×), car_racing (1×)
FK missing column	Dev	0	—
FK missing column	Train	5	baseball_1, imdb, loan_1, restaurants, store_product
FK target not key	Test	0	—
FK target not key	Dev	2	car_1, voter_1
FK target not key	Train	41	baseball_1 (19×), dorm_1 (3×), imdb (6×), soccer_1 (4×), wine_1 (2×), yelp (7×)

Table A3. Databases with empty tables in Spider Train partition

Database ID	Total tables	Empty tables	Data coverage
academic	15	15	0%
imdb	16	16	0%
yelp	7	7	0%
restaurants	3	3	0%
geo	7	7	0%
music_2	7	7	0%
scholar	10	10	0%
sakila_1	16	4	75%
formula_1	13	2	85%
Total (Train only)	94	71	24.5%

Table A4. Mixed consensus breakdown (percentages relative to Mixed category)

Disagreement pattern (Gemini-2.5-pro → GPT-5)	Spider test	Spider dev	Spider train
Correct → Partially Correct	315 (60.3%)	144 (59.5%)	1,375 (59.4%)
Correct → Incorrect	124 (23.8%)	57 (23.6%)	449 (19.4%)
Incorrect → Partially Correct	35 (6.7%)	17 (7.0%)	169 (7.3%)
Incorrect → Correct	20 (3.8%)	14 (5.8%)	121 (5.2%)
Correct → Unanswerable	14 (2.7%)	4 (1.7%)	94 (4.1%)
Partially Correct → Correct	6 (1.1%)	1 (0.4%)	11 (0.5%)
Partially Correct → Incorrect	6 (1.1%)	2 (0.8%)	30 (1.3%)
Incorrect → Unanswerable	1 (0.2%)	3 (1.2%)	40 (1.7%)
Unanswerable → Correct	0 (0.0%)	0 (0.0%)	14 (0.6%)
Unanswerable → Partially Correct	0 (0.0%)	0 (0.0%)	6 (0.3%)
Partially Correct → Unanswerable	0 (0.0%)	0 (0.0%)	5 (0.2%)
Unanswerable → Incorrect	1 (0.2%)	0 (0.0%)	2 (0.1%)

Table A5. Critical databases requiring immediate remediation

Database ID	Partition	Critical issues
sakila_1	Train	38,273 FK data violations + 4 empty tables
flight_2	Dev	2,400 FK data violations → EVALUATION BIAS
flight_4	Train	1,240 FK data violations
hr_1	Train	6 FK data violations
college_1	Train	2 FK data violations
car_1	Dev	2 FK data violations + structural FK errors
hospital_1	Train	1 FK data violation
allergy_1	Train	1 FK data violation
wta_1	Dev	1 FK data violation
pilot_1	Test	1 FK data violation → EVALUATION BIAS
book_1	Test	2 FK missing column
car_racing	Test	1 FK missing column
baseball_1	Train	FK missing column (Critical) + 19 FK target-not-key errors
imdb	Train	FK missing column (Critical) + 6 FK target-not-key errors + all 16 tables empty
loan_1	Train	1 FK missing column
store_product	Train	FK missing column
restaurants	Train	FK missing column (Critical) + all 3 tables empty

Authors and affiliations

Volodymyr Sabadosh
e-mail: volodymyr.sabadosh@uzhnu.edu.ua
ORCID: https://orcid.org/0009-0006-9933-444X
Uzhhorod National University, Ukraine

Vladyslav Kotsovsky
e-mail: vladyslav.kotsovsky@uzhnu.edu.ua
ORCID: https://orcid.org/0000-0002-7045-7933
Uzhhorod National University, Ukraine